67-272 Exam

NOT UPDATED

Note: This is not updated. I will try to include more for the Blocks section and the RegEx Section

Study Guide

EXAM DETAILS:

20% Design Issues and Subjects
20% General Ruby
20% Building Blocks
20% Regular Expressions

Design & Stuff

What is Design About?

- Simplicity
- Elegancy
- Aesthetics

But it is really About Tradeoffs. There is no perfect design

It's all about making tradeoffs

Apple iPhone:

Self Discoverable

The application does not need a manual

Google:

Really easy to use

Designed for what the user is looking for

Design catered toward user's interests and goals

Understanding Users

How do they go about doing their work?

Under what conditions do they have to work?

Small screens, environment, screen size, time constraints, etc.

Is there anything that can be done to simplify tasks?

What can we do to take advantage of the computer to extend functionality and capabilities?

What mistakes are they prone to making?

How can we limit mistakes?

What are their abilities and limitations?

Who are the users and what their abilities are

Get begging users to intermediate and advance levels ASAP

Who are the users?

Maybe not what we expect

Different types of users

Doing different tasks

Use Google analytics or other Analysis tools to find out more information on the users.

Location, times, technology used, etc.

Must spend time with users

Find out how they interact with the application

Understand their perspective and mental model

Learn from the User

Documenting analysis and findings

Put together profiles

Different types of profiles and categories of users

Cater test towards profiles to focus on most appropriate users

Use Cases

Actors

Different Types of Users

Levels

A, B, C:

A level Use Cases are required for a successful application

B level Use Cases are the very nice additions but not critical

C level Use Cases are nice but create flexibility in development

Only if resources (time, money) are left

Separate Use Cases

List all the use cases with an Actor, Level, and description (and maybe additional info)

Develop Test cases based on the Actors and Use Cases

Know what is acceptable and what is not acceptable

Test for extreme cases

Test the things you would never expect a user (considered to be a human) to do.

Try to break the system

Idiot/User Proof

Mental Models

Mental Models influence User's behavior.

Do not make ANY assumptions about a user's mental model

Mental Model:

The way a user thinks something works

How it works in their mind

This is all most always different from reality (maybe a lot of a little)

Correct Mental Models => Correct User Behavior

Create mappings for the user

Give the user clear directions/instructions/guides

The "Gulf of Execution"

The difference between what the User "Wants" to do and what the User "Can" do.

You want to save a file but the "Save" command is grayed out… wtf?

Sometime you need to limit the user from what they want to do because they are going to mess things up

The "Gulf of Evaluation"

What the user “Thinks” they did and what the user “Actually did”

One “Key” to narrowing these gulfs…

Tradeoffs

Know your tradeoffs, know where to expect the problems

Feedback

Get good user feedback to limit the “gulfs”

Error Messages

Understand who is going to see the Error Messages

Accurate and Understandable Error Message (Helpful Messages)

Clear action path

Do I click “OK” or “Cancel”?

I can’t tell from the given message…

Know the situations of your possible errors.

Other kinds of feedback

What events need feedback

Different means of providing feedback

Dialog boxes interrupt users

Use Color

Use Sounds

User other types of movement

Look at Google: after you send an email “email sent”, “3 emails deleted. Undo?” etc

Where is the system knowledge kept?

GUI vs. Command Line

A command line requires the user to have a lot of system knowledge in the user’s head.

Knowledge: In System vs. In Head

	In System	**In Head**
Retrievability	Whenever visible or audible	Requires memory search or reminder
Learning	Not-required – interpretation substitutes for learning	Required and can be considerable
Efficiency	Slowed by need to find and interpret external information	Can be very efficient
Aesthetics	Maintaining lots of information can lead to cluttered designs	Avoid cluttered look; simplicity can be aesthetically pleasing
Ease of Use (1st time)	High	Low

Golden Rules

Keep Users in Control

Give them freedom to do what they need to do

Reduce Memory Load (user's memory)

Visual Aids, Reminders, Guides

Keep interface Consistent

Keep structure and process consistent (as much as possible)

Reduces Memory Load

Wireframes

All Black & White

Colors distract

Only want to focus on functionality and basic layout

No logos or images (just placeholders)

Need to get the basics down before writing code

It is very expensive and go back and change the basics (foundation)

Easier to change appearance (Cheaper)

Show structure, layout, navigation, general makeup, functionality

Take the Use Cases and create a Visual Reference

(You don't have a lot of appearance with Use Cases)

Navigation Flow Charts

Demonstrate to the user (using the wireframes) the expected navigation and the functionality available at the different spots (so they can change things where they need to be changed ... early in development)

- No Graphics, Colors (this distracts the user from the main goal of this development phase.
- All Pages (create all the pages which demonstrate navigation and functionality)
- Putting in Storyboards (Put down the scenario the wireframe is depicting, note difference in scenarios => how the system changes for different situations)

Visual Design & Layout

1/3, 2/3

The Golden Ration

a/b = b/(a+b)

68% 32%

Main content = 68ish%

Left or Right column (32%)

Or split of the side(sub) columns (16%, 68%, 16%)

Colors

Basic Color Wheel

Different Colors communicate different things...

Green => relaxing

Red => Alerting, danger, concern

Orange => Lively Energetic

Yellow => Stimulating

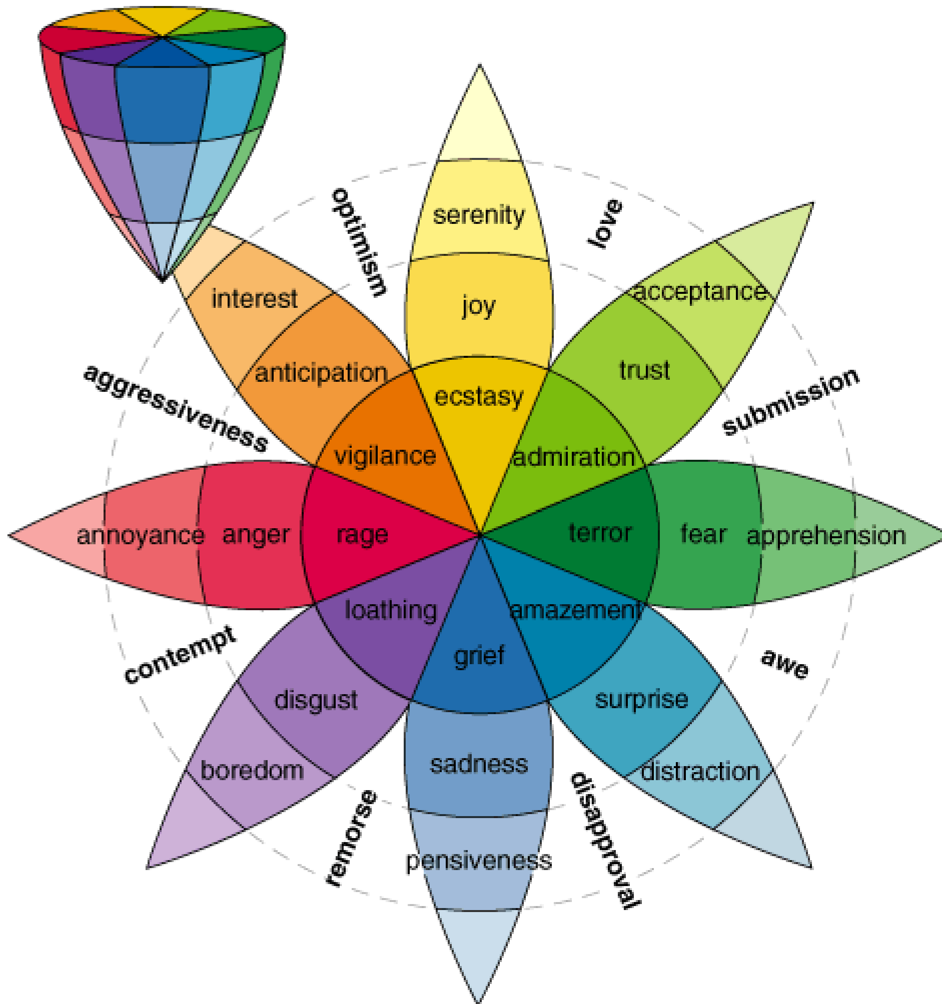
Purple => Rare, Rich, Luxurious, Wealth

Blue => Conservative, Calming, maybe depressing..?

Don't worry too much about the psychology

We need good Combinations

Find the right Combinations



Color Schemes

Monochromatic

White & Black do not count

All one color

Analogous

Choose a Color

And then choose two colors that are equal distance from blue

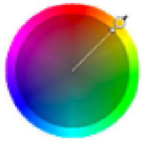
Complementary

Opposites on the Color Wheel

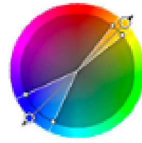
Breaking away…

You can sometimes break away from your color scheme to make certain things stand out

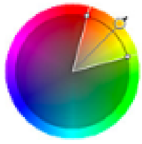
Color Schemes



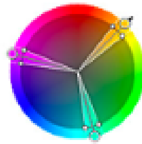
Monochromatic



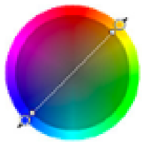
Split-Complementary



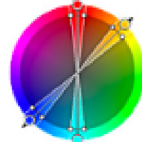
Analogous



Triadic



Complementary



Tetradic
(Double Complementary)

Typography

Serif Fonts vs. San-Serif Fonts

Son-serif for headings and title

Serif for main text with lots of word (more flowing)

Consider Conflicting or contrasting fonts (realize difference which may distract)

Too close and the conflict but if they are way too far about or have no relation they also distract

Funky fonts: Distracting, Ugly, hard to read, user may not have the fonts installed

Do not user too many fonts

Two fonts is Ideal

Three is an upper bound for a number of different fonts

A Serif and San-Serif

Use CSS to control layouts & typography (font family et.)

More on Design

Three Layers

- People
- Data
- And the interfaces that connect them

Obvious Design

- It conforms to the way users interact with the Web, but focuses on the activity instead of a specific audience.
- It has only those features that are absolutely necessary for users to complete the activity the application is meant to support.
- It supports the user's mental model of what it does.
- It helps user get started quickly so they can become intermediate users ASAP
- It makes it easy to recover from mistakes and difficult to make them in the first place.
- T has uniformly designed interface elements, but leverages irregularity to create meaning and importance.
- It reduces clutter to a minimum.

How to design the obvious

Turn the above "good qualities" into goals

Understand the user but also understand the activity

Remember the activity when select which features & functionality to offer

Do not build the Presentation and Appearance of the application off the technical implementation models (remember to support mental models)

Understand User Profiles and Tradeoffs between System Knowledge in Head and in System. Know the best way to get the user to intermediate/advanced ASAP. Do not make assumptions about this. Actually test and get feedback to see how the specific users of different profiles progress through this.

Consistency in design, structure & process allows a user to make less mistakes but you also have to account for the downright stupid...

Use "Use Cases" to create functional designs (requirements for the activity or task) but use User Feedback to design System Flow and Appearance (support User Mental Model and Stupidity)

The Framework for Obvious Design

Three parts:

- Know What to Build
- Know What Makes it Great
- Know the Best Ways to Implement it

What to Build

Purpose of the Application

Scope

Resources Available

Conceptual Element: What to build, not to build, underlying rationale for the app.

What Makes it Great

Great applications are composed of multiple aspects working together

Drag-and-drop vs. Drag-and-drop & feedback to communicate to user (mouse icon chance and real-time updating)

Know the Best Ways to Implement it

Interaction Element: enables the usability of an application and creates what many people call the "X-factor." This is the part of the application with which people actually interact with.

Users and your Application

Users latch onto the first application which they can use

The Change has to be substantial. Barriers to entry/competition. Switching Costs

Your Product/Application is not better with more features

Features may not be a competitive advantage. Many different factors to consider

Difficulty of accomplishing tasks in your application means that users don't stick around to fight their way through it, and they don't bother coming back.

Understand How Users Think They Do Things

Talking to a user about what they “would” do is different than watching them “actually do” something.

Just because you make something “too obvious” doesn’t mean the user will respond accordingly.

Understand How Users Actually Do Things

Many people are not Experts of Applications

Most use only 20% of an application functionality

Mental Models (more)

Different users use different applications for different purposes because of difference in mental models. So what’s the difference?

Use Surveys, shadowing, interview different user types(profiles) Assume Nothing, just get at reality

Design for the Activity

People adapt to technology so don’t bow down to them

Understand the task at hand and make it your primary focus. Design you application to allow for effective adaptation to the interface for accomplishing the task.

Build what is absolutely necessary

More Features, more frustration

Use Case **Priorities** related to the Activity, fact about **20%,** user will **reject app** if it is too complex to understand and accomplish task

Don’t get into a competitive battle that requires you to **introduce new features for new release to stay competitive … Think Different.**

Make sure the user can navigate to **80% of the application from the home screen**

Can use a variety of techniques to conserve screen space and reduce clutter while doing this: (Javascript to move things around or Hide/Show … Drop down menus etc)

Drop Unnecessary Tasks

Test:

Find features that do not contribute directly to the user's ability to complete a task that is vital to the activity the application is meant to support.

60 Second Deadline

"The project time line (resources) has been cut in half. We have 60 seconds to decide what to keep and what to throw away before we meet with the client in the conference room"

Once you complete the list of what you would now offer to the client, you can just start form that list to start developing your application.

Use Dashboards to conserve screen space and present user with all the relevant information

The Three R's

- **Requirements**:
  - Only the necessary requirements (Use Cases, Priorities, Resources)
- **Reduction**:
  - Reduce interfaces to their core as much as possible. Reduce the clutter, redundancy, the possibility of user error, reduce verbiage to the shortest possible and reduce.
- **Regularity**:
  - Make things look like you did things intentionally. Lining up input fields, consistent design, same spacing, typography that goes together.

Prototype the Design

Wire Frames

Paper Prototypes, HTML, click-through mockups (navigation flows) and Flash

Browserless test: Hide all the Browser tools/features and make sure you can accomplish you activity.

Getting Up to Speed

Turning users into intermediate fast.

Getting Started guide

Default Start page

Helpful tips on start up

Info icons which display help text on rollover

Wizards

Common Conventions (Design Patterns)

Instructive hints (grayed sample text in forms)

Instant Feedback (Ajax, javascript updates and feedback)

Do not wait until the user submits to check...

Choose Good Defaults

Helpful Documents

Design for Information

Known item: users know what they are looking for...

Exploratory: know what they want but can't translate for searching

Don't know what you need to do: user thinks they need A when they really need B.

Re-finding: user is attempting to find something they already have.

Mark favorite pages

Customize settings, displays, etc.

The 5 S Approach

Seri(**Sort**)

Sort through tools and other materials to determine what must be retained and what can be thrown out or stored.

Seiton(**Straighten**)

Arrange things into their most efficient and accessible arrangements.

Seiso(**Shine**)

Keep clean, tidy workplaces. Cleaning should be a frequent activity and should always be aimed at polishing up anything losing its shine.

Seiketsu(**Standardize**)

Leverage standards to enable consistency.

Online, adhering to standards means using proper semantic markup in Web pages and keeping the code used for presentation and content clearly separated.

Shitsuke(**Sustain**)

Sustain the work of each of the elements in the 5S system for the long haul.

Just-in-Time Design

The process of acquiring and delivering materials right when they're needed, as opposed to maintaining a surplus. In web terms, this translates to JIT Design which involves doing design work right after someone decided to add something to the interface and just before the programmer starts coding.

Extreme Programming

One Programmer, One Designer, One User, One Whiteboard

Ruby Part 1

Data types (examples)

```
# File to demo the use of Ruby Data Types

puts "Data Type Demonstration in Ruby"

# Numbers
# Fixnums & Bignums
# if the number is not a fixnum it is a bignum…

puts "123 is a #{123.class}"
puts "1234567890 x 1234567890 is a #{x = 1234567890 * 1234567890; x.class}"
puts

puts "1_000_000_000 is #{1_000_000_000}"
puts

puts "0377 is #{0377.to_f} in Octal"
puts "0b1111_1111 is #{0b1111_1111.to_f} in Binary"
puts "0xFF is #{0xFF.to_f} in Hexadecimal"
puts
puts

puts "Now some floating point literals"
#Numeric is the superclass of Integer, Float, Complex, BidDecimal, and Rational
# Numeric is the superclass of Fixnum  Bignum
puts 0.0.class
puts -3.14.class
puts "6.02e23 is a #{6.02e23.class} which equals #{6.02e23.to_i}"
puts
puts "Exponents"
puts "2**4 is #{2**4}"
puts "2**-1 is #{2**-1}"
puts "2**(1/3.0) is #{2**(1/3.0)}"
puts "2**(1/4) is (integer division) and equals #{2**(1/4)}"
puts "2**(1.0/4.0) is #{2**(1.0/4.0)}"
puts
puts "String Strings #{'Strings'.class}s"
myString = 'this is my string\'s class'
puts "this is my string\'s class => myString"
puts "the class of myString is #{myString.class}"
#Print with the first letter uppercase
puts myString.capitalize
#now change the first letter so it is uppercase
## This is called mutable…. you know, the opposite of java strings
## you can change the values of the string without creating new objects…
myString.capitalize!
puts myString
```

```
puts myString.upcase
puts "now Chop 5 times"
puts myString.chop.chop.chop.chop.chop
puts "but aren't you glad we didn't use chop!"
puts "using the \"!\" is called a \"destructive method \" "
puts "aren't you glad we can \"escape\" the \"'s"
puts myString
puts myString.empty?
puts myString.reverse
## Single vs Double Qoute Text/String
### Some difference in escaping with backslashing and newlineing
### Double qoutes support \n(newline), \t (tab), the #{evaluate_this}
puts "360 degrees=#{2*Math::PI} radians "
$salutation = 'hello'
puts "#$salutation world"
puts
puts
puts 'Now for some Arrays'
puts
puts [1,2,3].to_s
#just stores two values
puts [-10...0, 5..10,]
#store the number 1-100 into an array and then pring that array
puts (1..100).to_a
# array of subarrays
puts [[1,2,3],[1,2],[1,2,3,4]]
words = %w[this is a test]
puts words
puts Array.new.size
#pritns on each line
puts Array.new(3)
puts Array.new(4,0)
#.to_s combines into a string and then prints out the string
puts Array.new(3).to_s
puts Array.new(4,0).to_s
array = Array.new(3) {|i| i+1}
puts array
puts array.to_s
#index 2
puts array[2]
# (negative number) count from back
# Last element
puts array[-1]
# third to last element
puts array[-3]
a = ('a'..'z').to_a
puts a
puts a.size
# access with starting index and number of positions to include
puts a[0,0] # subarray has zero elements
puts a[1,1] # a one-element array
puts a[-3,3].to_s # the last three as a single string "xyz"
```

```
## now use a similar method to change the array
a[0,2] = ['A', 'B'] # Change first two elements to A and B
a[2...5] = ['x', 'r', 'e'] # change elements 2 3 4
puts a.to_s
#not sure what is up with the below
puts 2...5
puts 2..5
puts
#some more with changing arrays
a = [1,2,3] + [4,5] #[1,2,3,4,5]
a = a + [[6,7,8]] #[1,2,3,4,5, [6,7,8]]
# a = a + 9 =>> Error. right side must be an array
a = []
puts a.empty?
a << 1 # [1]
a << 2 << 3 # [1,2,3]
a << [4,5,6] # [1,2,3,[4,5,6]]
a = [ 1, 2, 3 ] * 3    #=> [ 1, 2, 3, 1, 2, 3, 1, 2, 3 ]
puts a.to_s
puts [ 1, 2, 3 ] * ","  #=> "1,2,3"
# Comparison—Returns an integer (-1, 0, or +1) if this array is less than, equal to, or greater than other_array.
puts [ "a", "a", "c" ]    <=> [ "a", "b", "c" ]   #=> -1
puts [ 1, 2, 3, 4, 5, 6 ] <=> [ 1, 2 ]            #=> +1

# Invokes block once for each element of self. Creates a new array containing the values returned by the block
a = [ "a", "b", "c", "d" ]
a.collect {|x| x + "!" }   #=> ["a!", "b!", "c!", "d!"]
puts a.to_s                #=> ["a", "b", "c", "d"]

# more on blocks later

# hash/maps/associative arrays ... whatever you want to call it
# maps a Key to a Value... Key Value Pairs..

# check out the different ways you can create a hash
hash1 = Hash.new
hash2 = []
hash3 = {}

# check out the different ways you can add key, value pairs to a hash
hash1["a"] = "firstletter"
hash1['b'] = "secondletter"
hash1[:c] = "thirdletter"

# notice how everything prints out the same
hash1.each {|key, value| puts "#{key} => #{value}"}
# a => firstletter
# c => thirdletter
# b => secondletter

# now check out a different way to create a new hash
# with different ways of setting key, value pairs
```

```
hash2 = {:one => 'numberone', "two" => 'numbertwo', 'three' => "numberthree"}

hash2.each {|key, value| puts "#{key} => #{value}"}

# three => numberthree
# two => numbertwo
# one => numberone

puts hash3.size # 0, empty hash

# check out "each" above...
# the term would be internal iterator (GoF)
# but you would be better off saying method that expects a block
# each (as does other iterators) "yields" some values
# in this case, "each" iterats through the hash and "yields" two parameters
# it yields the "key" and the "value"

# so let's see what happens when we don't account for a yeild...
# aka we don't pass a block that cathes/uses a parameter(ish) thing

hash2.each { puts "hello"} # Hello is printed 3 times
#how about
hash2.each { |i| puts i}
# prints
# three
# numberthree
# two
# numbertwo
# one
# numberone

# can you see what "each" is doing?
# it is going through and yielding certain values to the block
# "each" yields two paramters on each iteration
# "each" yields the "key" and then the "value"

# so we need to construct our block as follows
hash2.each {|i, j| puts "#{i} => #{j}"}

#or an alternative

hash2.each  do |i, j|
  puts "#{i} => #{j}"
end

#same output

#Ranges

range = 1..10
range.each {|i| puts i} # => prints 1 though 10 on newlines
```

```
puts range.include? 2  # => true

range = 10...20 # DOES include 10... doesNOT include 20
range.each {|i| puts i} # prints 10 through 19 (not 20)
# you can see that the range object's method "each" returns/yields one value

#you can "step" through with a declared number of advancements
range.step(2) {|i| puts i} # pritns 10,12,14,16,18
range.step(3) {|j| puts j} # prints 10,13,16,19

# repition
3.times {puts "hello"} # prints hello three times
3.times puts "hello" # prints hello ONCE
# above, the 3.times is not associated with puts
# must use the brackets like the first one
```

Ruby Part 2

Things to know…

Meta Programming

From Wikipedia"

***Metaprogramming** is the writing of computer programs that write or manipulate other programs (or themselves) as their data, or that do part of the work at runtime that would otherwise be done at compile time. In many cases, this allows programmers to get more done in the same amount of time as they would take to write all the code manually, or it gives programs greater flexibility to efficiently handle new situations without recompilation.*

The language in which the metaprogram is written is called the metalanguage. The language of the programs that are manipulated is called the object language. The ability of a programming language to be its own metalanguage is called reflection or reflexivity.

Reflection is a valuable language feature to facilitate metaprogramming. Having the programming language itself as a first-class data type (as in Lisp or Rebol) is also very useful. Generic programming invokes a metaprogramming facility within a language, in those languages supporting it.

Metaprogramming usually works through one of two ways. The first way is to expose the internals of the run-time engine to the programming code through application programming interfaces (APIs). The second approach is dynamic execution of string expressions that contain programming commands. Thus, "programs can write programs". Although both approaches can be used in the same language, most languages tend to lean toward one or the other.

Or try this reference: http://www.vanderburg.org/Speaking/Stuff/oscon05.pdf

Reserved words

```
alias   and     BEGIN   begin   break   case    class   def     defined?
do      else    elsif   END     end     ensure  false   for     if
in      module  next    nil     not     or      redo    rescue  retry
return  self    super   then    true    undef   unless  until   when
while   yield
```

Strings

In all of the %() cases below, you may use any matching characters or any single character for delimiters. %[], %!!, %@@, etc.

```
'no interpolation'
"#{interpolation}, and backslashes\n"
%q(no interpolation)
%Q(interpolation and backslashes)
%(interpolation and backslashes)
`echo command interpretation with interpolation and backslashes`
%x(echo command interpretation with interpolation and backslashes)
```

Backslashes

```
\t (tab), \n (newline), \r (carriage return), \f (form feed), \b
(backspace), \a (bell), \e (escape), \s (whitespace), \nnn (octal),
\xnn (hexadecimal), \cx (control x), \C-x (control x), \M-x (meta x),
\M-\C-x (meta control x)
```

Symbols

Internalized String. Guaranteed to be unique and quickly comparable. Ideal for hash keys. Symbols may not contain \0 or be empty.

```
:symbol                          => :symbol
:'#{"without"} interpolation'  => :"#{"without"} interpolation"
:"#{"with"} interpolation"     => :"with interpolation"
%s(#{"without"} interpolation) => :"#{"without"} interpolation"
```

Arrays

```
[1, 2, 3]
%w(foo bar baz)
%W(foo bar baz #{var})
```

Indexes may be negative, and they index backwards (eg -1 is last element).

Variables

```
$global_variable
@@class_variable
@instance_variable
[OtherClass::]CONSTANT
local_variable
```

Invoking a Method

Nearly everything available in a method invocation is optional, consequently the syntax is very difficult to follow. Here are some examples:

- *method*
- *obj.method*
- *Class::method*
- *method(key1 => val1, key2 => val2)*
  - *is **one** argument for def method(hash_arg) !*
- *method(arg1, *[arg2, arg3]) becomes: method(arg1, arg2, arg3)*

- *as ugly as you want it to be:*
    - *method(arg1, key1 => val1, key2 => val2, *splat_arg) #{ block }*

```
invocation := [receiver ('::' | '.')] name [ parameters ] [ block ]
parameters := ( [param]* [, hashlist] [*array] [&aProc] )
block      := { blockbody } | do blockbody end
```

Defining a Class

Class names begin w/ capital character.

```
class Identifier [< superclass ]
  expr..
end
# singleton classes, add methods to a single instance
class << obj
  expr..
end
```


Defining a Class

Class names begin w/ capital character.

```
class Identifier [< superclass ]
  expr..
end
# singleton classes, add methods to a single instance
class << obj
  expr..
end
```


Defining a Method

```
def method_name(arg_list, *list_expr, &block_expr)
  expr..
end
# singleton method
def expr.identifier(arg_list, *list_expr, &block_expr)
  expr..
end
```

- *All items of the arg list, including parens, are optional.*
- *Arguments may have default values (name=expr).*
- *Method_name may be operators (see above).*
- *The method definitions can not be nested.*
- *Methods may override operators: .., |, ^, &, <=>, ==, ===, =~, >, >=, <, <=, +, -, *, /, %, **, <<, >>, ~, +@, -@, [], []= (2 args)*

Defining a Method

```
def method_name(arg_list, *list_expr, &block_expr)
```

```
  expr..
end
# singleton method
def expr.identifier(arg_list, *list_expr, &block_expr)
  expr..
end
```

- *All items of the arg list, including parens, are optional.*
- *Arguments may have default values (name=expr).*
- *Method_name may be operators (see above).*
- *The method definitions can not be nested.*
- *Methods may override operators: .., |, ^, &, <=>, ==, ===, =~, >, >=, <, <=, +, -, *, /, %, **, <<, >>, ~, +@, -@, [], []= (2 args)*

Accessors

Class Module provides the following utility methods:

attr_reader <attribute>[, <attribute>]...

Creates a read-only accessor for each <attribute>.

attr_writer <attribute>[, <attribute>]...

Creates a write-only accessor for each <attribute>.

attr <attribute> [, <writable>]

Equivalent to "attr_reader <attribute>; attr_writer <attribute> if <writable>"

attr_accessor <attribute>[, <attribute>]...

Equivalent to "attr <attribute>, TRUE" for each argument.

Aliasing

```
alias        :new  :old
alias_method :new, :old
```

Creates a new reference to whatever old referred to. old can be any existing method, operator, global. It may not be a local, instance, constant, or class variable.

Blocks, Closures, and Procs

Blocks/Closures

- *blocks must follow a method invocation:*

```
invocation do ... end
invocation { ... }
```

- *Blocks remember their variable context, and are full closures.*
- *Blocks are invoked via yield and may be passed arguments.*
- *Brace form has higher precedence and will bind to the last parameter if invocation made w/o parens.*
- *do/end form has lower precedence and will bind to the invocation even without parens.*

Proc Objects

Created via:

- *Kernel#proc*
- *Proc#new*
- *By invoking a method w/ a block argument.*

See class Proc for more information.

Raising and Rescuing

```
raise ExceptionClass[, "message"]
begin
  expr..
[rescue [error_type [=> var],..]
  expr..]..
[else
  expr..]
[ensure
  expr..]
end
```

Class Hierarchy

- *Object*
    - *Hash*
    - *Symbol*
    - *IO*
        - *File*
    - *Continuation*
    - *File::Stat*
    - *Data*
    - *NilClass*
    - *Exception (see tree above)*
    - *Array*
    - *Proc*
    - *String*
    - *Numeric*
        - *Float*
        - *Integer*
            - *Bignum*
            - *Fixnum*
    - *Regexp*
    - *Thread*
    - *Module*
        - *Class*
    - *ThreadGroup*
    - *Method*
        - *UnboundMethod*
    - *Struct*
        - *Struct::Tms*
    - *TrueClass*
    - *Time*
    - *Dir*
    - *Binding*
    - *Range*

- *MatchData*
- *FalseClass*

irb

```
irb [options] [script [args]]
```

The essential options are:

```
-d               Sets $DEBUG to true. Same as "ruby -d ..."
-f               Prevents the loading of ~/.irb.rc.
-h               Get a full list of options.
-m               Math mode. Overrides --inspect. Loads "mathn.rb".
-r module        Loads a module. Same as "ruby -r module ..."
-v               Prints the version and exits.
--inf-ruby-mode Turns on emacs support and turns off readline.
--inspect        Turns on inspect mode. Default.
--noinspect      Turns off inspect mode.
--noprompt       Turns off the prompt.
--noreadline     Turns off readline support.
--prompt         Sets to one of 'default', 'xmp', 'simple', or 'inf-ruby'.
--readline       Turns on readline support. Default.
--tracer         Turns on trace mode.
```

Command Line Options

```
-0[octal]        specify record separator (\0, if no argument).
-a               autosplit mode with -n or -p (splits $_ into $F).
-c               check syntax only.
-Cdirectory      cd to directory, before executing your script.
--copyright      print the copyright and exit.
-d               set debugging flags (set $DEBUG to true).
-e 'command'     one line of script. Several -e's allowed.
-F regexp        split() pattern for autosplit (-a).
-h                prints summary of the options.
-i[extension]    edit ARGV files in place (make backup if extension supplied).
-Idirectory      specify $LOAD_PATH directory (may be used more than once).
-Kkcode          specifies KANJI (Japanese) code-set.
-l               enable line ending processing.
-n               assume 'while gets(); ... end' loop around your script.
-p               assume loop like -n but print line also like sed.
-rlibrary        require the library, before executing your script.
-s               enable some switch parsing for switches after script name.
-S               look for the script using PATH environment variable.
-T[level]        turn on tainting checks.
-v               print version number, then turn on verbose mode.
--version        print the version and exit.
-w               turn warnings on for your script.
-x[directory]    strip off text before #! line and perhaps cd to directory.
-X directory     causes Ruby to switch to the directory.
-y               turns on compiler debug mode.
```

http://www.zenspider.com/Languages/Ruby/QuickRef.html#2

Regular Expressions

regex characters:

```
.              any character except newline
[ ]            any single character of set
[^ ]           any single character NOT of set
*              0 or more previous regular expression
*?             0 or more previous regular expression (non-greedy)
+              1 or more previous regular expression
+?             1 or more previous regular expression (non-greedy)
?              0 or 1 previous regular expression
|              alternation
( )            grouping regular expressions
^              beginning of a line or string
$              end of a line or string
{m,n}          at least m but most n previous regular expression
{m,n}?         at least m but most n previous regular expression (non-greedy)
\1-9           nth previous captured group
\&             whole match
\`             pre-match
\'             post-match
\+             highest group matched
\A             beginning of a string
\b             backspace(0x08)(inside[]only)
\b             word boundary(outside[]only)
\B             non-word boundary
\d             digit, same as[0-9]
\D             non-digit
\S             non-whitespace character
\s             whitespace character[ \t\n\r\f]
\W             non-word character
\w             word character[0-9A-Za-z_]
\z             end of a string
\Z             end of a string, or before newline at the end
(?#)           comment
(?:)           grouping without backreferences
(?=)           zero-width positive look-ahead assertion
(?!)           zero-width negative look-ahead assertion
(?>)           nested anchored sub-regexp. stops backtracking.
(?imx-imx)     turns on/off imx options for rest of regexp.
(?imx-imx:re)  turns on/off imx options, localized in group.
```

special character classes:

```
[:alnum:]   alpha-numeric characters
[:alpha:]   alphabetic characters
[:blank:]   whitespace - does not include tabs, carriage returns, etc
[:cntrl:]   control characters
[:digit:]   decimal digits
[:graph:]   graph characters
[:lower:]   lower case characters
[:print:]   printable characters
[:punct:]   punctuation characters
[:space:]   whitespace, including tabs, carriage returns, etc
[:upper:]   upper case characters
[:xdigit:]  hexadecimal digits
```